A Private Lookup Protocol with Low Online Complexity for Secure Multiparty Computation

Peeter Laud

Cybernetica AS
peeter.laud@cyber.ee

Abstract. We present a secure multiparty computation (SMC) protocol for obliviously reading an element of an array, achieving constant *online* communication complexity. While the total complexity of the protocol is linear in the size of the array, the bulk of it is pushed into the offline precomputation phase, which is independent of the array and the index of the element.

Although private lookup is less general than oblivious RAM (ORAM), it allows us to give new and/or more efficient SMC protocols for a number of important computational tasks. In this paper, we present protocols for executing deterministic finite automata (DFA), and for finding shortest distances in sparse graphs.

All our protocols are given in the arithmetic black box model, which allows them to be freely composed and used in larger applications.

Keywords: secure multiparty computation; arithmetic black box; private lookup

1 Introduction

In Secure Multiparty Computation (SMC), p parties compute $(y_1, \ldots, y_p) = f(x_1, \ldots, x_p)$, with the party P_i providing the input x_i and learning no more than the output y_i. For any functionality f, there exists a SMC protocol for it [36, 19]. While the general construction is inefficient in practice, several SMC frameworks have appeared [12, 3, 6, 21, 29] and certain classes of algorithms can be executed with reasonable efficiency on top of them. In particular, these algorithms should have control flow and data access patterns that depend only or mostly on public data.

Private information retrieval (PIR) and oblivious RAM (ORAM) are among techniques for hiding the patterns of data access. They both posit a client-server setting, where the client queries the elements in server's memory without the server learning which elements are accessed. For a n-element vector, the asymptotic complexity of both PIR (which allows only reading) and ORAM techniques (which also allow writing) is $\tilde{O}(\log^2 n)$. To adapt these techniques to the SMC setting, at least the client's computations have to be performed through SMC protocols. This brings further complications.

We provide an alternative mechanism for reading an element of a vector according to a private index (private writing is not considered in this paper). We give a private lookup protocol, the operations of which can be partitioned into the offline part — these that can be done without knowing the actual inputs —, and the online part — these that require the inputs. In case of private lookup, it makes sense to consider even three phases — offline, vector-only (where the actual vector, either public or private, is available) and online (where the private index is also available). In our protocols, the online phase requires only a constant number of costly SMC operations, while the bulk of the work is done in the offline and, depending on the protocol and the secrecy of the vector elements, in the vector-only phases. In cases where the main cost of SMC operations is communication between parties, the offline (and vector-only) computations could be performed using dedicated high-bandwidth high-latency channels.

Our private lookup protocols are universally composable, they may be freely used as components in protocols for more complex privacy-preserving applications. In this paper we demonstrate their use in two applications that need oblivious read access to data, but where the pattern for write accesses is public. We have implemented SMC protocols for executing deterministic finite automata (DFA), and for finding the single-source shortest distances (SSSD) in sparse graphs. The latter protocol is based on the well-known Bellman-Ford algorithm. In both protocols, the processed objects (automaton, input string, the graph) are private, except for their sizes.

Our protocols inherit the security guarantees of the underlying SMC implementation. If the SMC implementation provides security against passive resp. also active adversaries, then so do our protocols. If the security provided by the SMC implementation is information-theoretical resp. only computational, then this also applies to our protocols. Perhaps surprisingly, the protocols in this paper are the first *information-theoretically* secure protocols for DFA execution, if an information-theoretically secure protocol set for SMC is used. All previous protocols have used cryptographic constructions (encryption) that rely on computational hardness assumptions for security.

Structure of the paper We review work related to privacy-preserving data access, and to our example applications in Sec. 2. In Sec. 3 we describe the framework in which our protocols are defined and their security and performance properties stated. Sec. 4 presents the private lookup and discusses its security and performance. In Sections 5 and 6 we describe our implementations of privacy-preserving DFA execution and SSSD, and discuss their performance. Finally, we draw the conclusions in Sec. 7.

2 Related work

Secure multiparty computation (SMC) protocol sets can be based on a variety of different techniques, including garbled circuits [36], secret sharing [32, 17, 4] or homomorphic encryption [9]. A highly suitable abstraction of SMC is the universally composable Arithmetic Black Box (ABB) [14], the use of which allows

very simple security proofs for higher-level SMC applications. Using the ABB to derive efficient privacy-preserving implementations for various computational tasks is an ongoing field of research [11, 8, 1, 27], also containing this paper.

Conceptually, our protocols are most similar to private information retrieval (PIR) [24], for which there exist protocols with $O(\log^2 n)$ communication and $O(n/\log n)$ public-key operations [26]. We know of no attempts to implement these techniques on top of SMC, though.

Oblivious RAM (ORAM) [20] is a more versatile technique with similar communication complexity [33] (but higher round complexity and client's memory requirements). The integration of ORAM with SMC has been studied [13, 22, 18, 28]. In general, such systems require at least $O(\log^3 n)$ overhead for oblivious data access. We also note that the "trivial" way of expanding the private index into its characteristic vector and computing its scalar product with the array brings $O(n)$ overhead, but may be more efficient in practice due to smaller constants hidden in the O-notation [25, 22].

In this paper, we present protocols for DFA execution, and for SSSD in sparse graphs. In [15, 30], garbled circuits have been adapted for DFA execution, where one party knows the DFA and the other one the input string. This approach, which is not universally composable, works well if the automaton and alphabet are small (but the input string may be long). In [34, 2, 16, 35], DFA execution protocols based on homomorphic encryption are given, some of them resembling PIR protocols. Privacy-preserving graph algorithms have been studied in [5] in a non-composable manner. Composable SSSD protocols for dense graphs have been studied in [1]. Recently, ORAM-with-SMC techniques have been used to implement Dijkstra's algorithm for sparse graphs [22].

3 Preliminaries

Universal composability (UC) [7] is a framework for stating security properties of systems. It considers an ideal functionality $\mathcal{F}$ and its implementation π with identical interfaces to the intended users. The latter is *at least as secure as* the former, if for any attacker $\mathcal{A}$ there exists an attacker $\mathcal{A}_S$, such that $\pi \| \mathcal{A}$ and $\mathcal{F} \| \mathcal{A}_S$ are indistinguishable to any potential user of π / $\mathcal{F}$. The value of the framework lies in the composability theorem: if π is at least secure as $\mathcal{F}$, then ξ^π is at least as secure as $\xi^\mathcal{F}$ for any system ξ that uses π / $\mathcal{F}$. We say that such ξ is implemented *in the $\mathcal{F}$-hybrid model*. When arguing about the security of such ξ, we may assume that it uses the ideal functionality $\mathcal{F}$ as a subroutine. All derived conclusions will be valid also for ξ^π.

The *arithmetic black box* is an ideal functionality $\mathcal{F}_{\mathsf{ABB}}$. It allows its users (a fixed number p of parties) to securely store and retrieve values, and to perform computations with them. When a party sends the command $\mathsf{store}(v)$ to $\mathcal{F}_{\mathsf{ABB}}$, where v is some value, the functionality assigns a new *handle* h (sequentially taken integers) to it by storing the pair (h, v) and sending h to all parties. If a sufficient number (depending on implementation details) of parties send the command $\mathsf{retrieve}(h)$ to $\mathcal{F}_{\mathsf{ABB}}$, it looks up (h, v) among the stored pairs and responds

with v to all parties. When a sufficient number of parties send the command $\mathsf{compute}(op; h_1, \ldots, h_k; params)$ to $\mathcal{F}_{\mathsf{ABB}}$, it looks up the values $v_1, \ldots, v_k$ corresponding to the handles $h_1, \ldots, h_k$, performs the operation op (parametrized with $params$) on them, stores the result v together with a new handle h, and sends h to all parties. In this way, the parties can perform computations without revealing anything about the intermediate values or results, unless a sufficiently large coalition wants a value to be revealed.

The existing implementations of ABB are protocol sets π_{ABB} based on either secret sharing [12, 3, 6] or threshold homomorphic encryption [14, 21]. Depending on the implementation, the ABB offers protection against a honest-but-curious, or a malicious party, or a number of parties (up to a certain limit). E.g. the implementation of the ABB by Sharemind [3] consists of three parties, providing protection against one honest-but-curious party.

In this paper, the protocols are given and their security (and correctness) argued in the $\mathcal{F}_{\mathsf{ABB}}$-hybrid model. The arguments remain valid if $\mathcal{F}_{\mathsf{ABB}}$ is replaced with a secure implementation π_{ABB}.

Typically, the ABB performs computations with values v from some ring $\mathbb{R}$. The set of operations definitely includes addition/subtraction, multiplication of a stored value with a public value (this operation motivates the *params* in the compute-command), and multiplication. Even though all algorithms can be expressed using just these operations, most ABB implementations provide more operations (as primitive protocols) for greater efficiency of the implementations of algorithms on top of the ABB. In all ABB implementations, addition, and multiplication with a public value occur negligible costs; hence they're not counted when analyzing the complexity of protocols using the ABB. Other operations may require a variable amount of communication (in one or several rounds) between parties, and/or expensive computation. The ABB can execute several operations in parallel; the *round complexity* of a protocol is the number of communication rounds all operations of the protocol require, when parallelized as much as possible.

It is common to use $[\![v]\!]$ to denote the value v stored in the ABB. The notation $[\![v_1]\!]$ *op* $[\![v_2]\!]$ denotes the computation of v_1 *op* v_2 by the ABB (translated to a protocol in the implementation π_{ABB}).

In the next section, we give a protocol for private lookup. Formally, we are presenting a secure implementation for the functionality $\mathcal{F}_{\mathsf{ABB+LU}}$ that accepts the same commands as $\mathcal{F}_{\mathsf{ABB}}$, answering them in the same manner. Additionally, it accepts the command $\mathsf{lookup}(h_1, \ldots, h_n, h_{\mathsf{idx}})$. When a sufficient number of parties has sent such command to $\mathcal{F}_{\mathsf{ABB+LU}}$, it looks up the value v_{idx} corresponding to h_{idx} and the value v' corresponding to $h_{v_{\mathsf{idx}}}$. It stores v' together with a new handle h' and sends h' to all parties.

The implementation $\pi_{\mathsf{ABB+LU}}$ is given in the $\mathcal{F}_{\mathsf{ABB}}$-hybrid model. It simply invokes $\mathcal{F}_{\mathsf{ABB}}$ for all $\mathcal{F}_{\mathsf{ABB}}$ commands. The implementation of the lookup-command is given below.

Algorithm 1: Private look-up protocol

Data: Vector of indices $i_1, \ldots, i_m \in \mathbb{F} \backslash \{0\}$
Data: Vector of values $([\![v_{i_1}]\!], \ldots, [\![v_{i_m}]\!])$ with $v_{i_1}, \ldots, v_{i_m} \in \mathbb{F}$.
Data: Index $[\![j]\!]$ to be looked up, with $j \in \{i_1, \ldots, i_m\}$.
Result: The looked up value $[\![w]\!] = [\![v_j]\!]$.

Offline phase

1 $([\![r]\!], [\![r^{-1}]\!]) \xleftarrow{\$} \mathbb{F}^*$
2 **for** $k = 2$ **to** $m - 1$ **do** $[\![r^j]\!] \leftarrow [\![r]\!] \cdot [\![r^{j-1}]\!]$;
3 Compute the coefficients $\lambda^{\mathbf{I}}_{j,k}$ from $i_1, \ldots, i_m$.

Vector-only phase

4 **foreach** $k \in \{0, \ldots, m-1\}$ **do** $[\![c_k]\!] \leftarrow \sum_{l=1}^{m} \lambda^{\mathbf{I}}_{k,l} [\![v_l]\!]$;
5 **foreach** $k \in \{0, \ldots, m-1\}$ **do** $[\![y_k]\!] \leftarrow [\![c_k]\!] \cdot [\![r^k]\!]$;

Online phase

6 $z \leftarrow \mathsf{retrieve}([\![j]\!] \cdot [\![r^{-1}]\!])$
7 $[\![w]\!] = \sum_{k=0}^{m-1} z^k [\![y_k]\!]$

4 Protocol for private lookup

Our protocol, depicted in Algorithm 1, takes the handles to elements $v_{i_1}, \ldots, v_{i_m}$ (with arbitrary non-zero, mutually different indices) and the handle to the index j stored inside the ABB, and returns a handle to the element v_j. It represents the elements as a polynomial V over a suitable field $\mathbb{F}$, satisfying $V(i_j) = v_{i_j}$ for all $j \in \{1, \ldots, m\}$ (with $i_1, \ldots, i_m$ also belonging to $\mathbb{F}$). The lookup then amounts to the evaluation of the polynomial in a point. Similar ideas have appeared in [35] (for DFAs). We will then combine these ideas with a method to move offline most of the computations for the polynomial evaluation [27]. Both the idea and the method have been slightly improved and expanded in this paper.

Let our ABB work with the values from the field $\mathbb{F}$, where $|\mathbb{F}| \geq m+1$. There exist protocols for generating a uniformly random element of $\mathbb{F}$ inside the ABB (denote: $[\![r]\!] \xleftarrow{\$} \mathbb{F}$), and for generating a uniformly random non-zero element of $\mathbb{F}$ together with its inverse (denote: $([\![r]\!], [\![r^{-1}]\!]) \xleftarrow{\$} \mathbb{F}^*$). These protocols require a small constant number of multiplications on average for any ABB [11].

There exist Lagrange interpolation coefficients $\lambda^{\mathbf{I}}_{j,k}$ depending only on the set $\mathbf{I} = \{i_1, \ldots, i_m\}$, such that $V(x) = \sum_{j=0}^{m-1} c_j x^j$, where $c_j = \sum_{k=1}^{m} \lambda^{\mathbf{I}}_{j,k} v_{i_k}$. These coefficients are public and computed in the offline phase of Alg. 1.

Correctness The definition of c_k gives $\sum_{k=0}^{m-1} c_k l^k = v_l$ for all $l \in \{i_1, \ldots, j_m\}$. We can now verify that $w = \sum_{k=0}^{m-1} y_k z^k = \sum_{k=0}^{m-1} c_k r^k j^k r^{-k} = v_j$.

Security and privacy To discuss the security properties of a protocol in the $\mathcal{F}_{\mathsf{ABB}}$-hybrid model, we only have to consider which extra information the adversary may be able to obtain from the retrieve-commands, and how it can affect the run of the protocol through the values it store-s (the latter is significant only if the adversary is active). There are no store-commands in Alg. 1. The results of the retrieve-commands are uniformly randomly distributed elements of

$\mathbb{F}^*$, independent of everything else the adversary sees. These can be simulated without any access to $v_{i_1}, \ldots, v_{i_m}$ and j. Hence Alg. 1 is secure and private against the same kinds of adversaries that the used implementation π_{ABB} of $\mathcal{F}_{\mathsf{ABB}}$ can tolerate.

Complexity In the offline stage, we perform $m-2$ multiplications. We also generate one random invertible element together with its inverse, this generation costs the same as a couple of multiplications [11]. The round complexity of this computation, as presented in Alg. 1 is also $O(m)$, which would be bad for online computations. For offline computations, the acceptability of such round complexity mainly depends on the latency of the used communication channels. The offline phase could be performed in $O(1)$ rounds [11] at the cost of increasing the number of multiplications a couple of times. In the vector-only phase, the computation of the values $[\![c_k]\!]$ is free, while the computation of the values $[\![y_k]\!]$ requires $m-1$ multiplications (the computation of $[\![y_0]\!]$ is free). All these multiplications can be performed in parallel. If the vector v were public then the computation of $[\![y_k]\!]$ would have been free, too. The only costly operations in the online phase is a single multiplication and a single retrieve-operation; these have similar complexities in existing ABB implementations.

4.1 Speeding up the offline phase

The preceding complexity analysis is valid for any implementation of the ABB. Some implementations contain additional efficient operations that speed up certain phases of Alg. 1. If we use the additive secret sharing based implementation, as used in Sharemind [4], and a binary field $\mathbb{F}$, then we can bring down the complexity of the offline phase to $O(\sqrt{m})$ as shown in the following.

The Sharemind ABB is realized by three parties, offering protection against passive attacks by one of the parties. The ABB stores elements of some ring $\mathbb{R}$; a value $v \in \mathbb{R}$ is represented by π_{ABB} as $[\![v]\!] = ([\![v]\!]_1, [\![v]\!]_2, [\![v]\!]_3) \in \mathbb{R}^3$ satisfying $[\![v]\!]_1 + [\![v]\!]_2 + [\![v]\!]_3 = v$, where the *share* $[\![v]\!]_i$ is kept by the i-th party P_i. Messages depending on these shares are sent among the parties, hence it is important to rerandomize $[\![v]\!]$ before each use. The *resharing* protocol [4, Algorithm 1] (repeated here as Alg. 2; all indices of the parties are *modulo* 3) is used for this rerandomization. We note that in this algorithm, the generation and distribution of random elements can take place offline. Even better, only random seeds can be distributed ahead of the computation and new elements of $\mathbb{R}$ generated from them as needed. Hence we consider the resharing protocol to involve only local operations and have the cost 0 in our complexity analysis.

If the ring $\mathbb{R}$ is a binary field $\mathbb{F}$, then the additive sharing is actually bit-wise secret sharing: $[\![v]\!] = [\![v]\!]_1 \oplus [\![v]\!]_2 \oplus [\![v]\!]_3$, where $\oplus$ denotes bit-wise exclusive or. For such sharings, the usual arithmetic operations with shared values in $\mathbb{Z}_{2^n}$ are more costly, compared to additive sharings over $\mathbb{Z}_{2^n}$, but equality checks and comparisons are cheaper [4]. As most operations with array indices are expected to be comparisons, bit-wise secret sharing may be a good choice for them.

Sharemind's multiplication protocol [4, Algorithm 2] (repeated as Alg. 3) is based on the equality $([\![u]\!]_1 + [\![u]\!]_2 + [\![u]\!]_3)([\![v]\!]_1 + [\![v]\!]_2 + [\![v]\!]_3) = \sum_{i,j=1}^{3} [\![u]\!]_i [\![v]\!]_j$.

Algorithm 2: Resharing protocol $[\![w]\!] \leftarrow \mathsf{Reshare}([\![u]\!])$ in SHAREMIND [4]

Data: Value $[\![u]\!]$
Result: Value $[\![w]\!]$ such that $w = u$ and the components of $[\![w]\!]$ are independent of everything else

Party P_i generates $r_i \xleftarrow{\$} \mathbb{R}$, sends it to party P_{i+1}
Party P_i computes $[\![w]\!]_i \leftarrow [\![u]\!]_i + r_i - r_{i-1}$

Algorithm 3: Multiplication protocol in the ABB of SHAREMIND [4]

Data: Values $[\![u]\!]$ and $[\![v]\!]$
Result: Value $[\![w]\!]$, such that $w = uv$

1 $[\![u']\!] \leftarrow \mathsf{Reshare}([\![u]\!])$
2 Party P_i sends $[\![u']\!]_i$ to party P_{i+1}
3 $[\![v']\!] \leftarrow \mathsf{Reshare}([\![v]\!])$
4 Party P_i sends $[\![v']\!]_i$ to party P_{i+1}
5 Party P_i computes $[\![w']\!]_i \leftarrow [\![u']\!]_i \cdot [\![v']\!]_i + [\![u']\!]_i \cdot [\![v']\!]_{i-1} + [\![u']\!]_{i-1} \cdot [\![v']\!]_i$
6 $[\![w]\!] \leftarrow \mathsf{Reshare}([\![w']\!])$

After the party P_i has sent $[\![u]\!]_i$ and $[\![v]\!]_i$ to party P_{i+1} (here and subsequently, all party indices are *modulo* 3), each of these nine components of the sum can be computed by one of the parties. The multiplication protocol is secure against one honest-but-curious party [4, Theorem 2]. Indeed, as the sending of $[\![u]\!]_i$ from P_i to P_{i+1} takes place after resharing $[\![u]\!]$, the value $[\![u]\!]_i$ is a uniformly random number independent of all other values P_{i+1} sees. Hence the simulator for P_{i+1}'s view could itself generate this value. The same consideration also underlies the security proof of the specialized offline phase protocol given in Alg. 4, the properties of which we discuss below.

Privacy We have to show that the view of a single party P_i can be simulated without access to the shares held by other parties. Party P_i receives messages only in lines 4 and 9 of Alg. 4. In both cases, it receives a share of a freshly reshared value. Hence this message can be simulated by a uniformly random number, as discussed above.

Complexity We consider local computation and resharings to be free, hence we have to count the number of messages sent by the parties. It is easy to see that a party sends at most $\sqrt{m+1}$ elements of the field $\mathbb{F}$ in lines 4 and 9 of Alg. 4. This is also the round complexity of Alg. 4. But by tracking the data dependencies in the first loop, we see that its iterations no. $2^{k-1}, \ldots, 2^k - 1$ could be done in parallel for each $k \in \{1, \ldots, q-1\}$. Hence Alg. 4 could be executed in $O(\log m)$ rounds.

Correctness We can use Alg. 4 only if $\mathbb{F}$ is a binary field. In this case squaring a shared value $[\![u]\!]$ is a local operation: $([\![u]\!]_1 + [\![u]\!]_2 + [\![u]\!]_3)^2 = [\![u]\!]_1^2 + [\![u]\!]_2^2 + [\![u]\!]_3^2$ and the computation of $[\![u^2]\!]_i = [\![u]\!]_i^2$ only requires the knowledge of $[\![u]\!]_i$. Regarding Alg. 4, note that its first loop satisfies the invariant that in the beginning of each iteration, each party P_i knows the values $[\![v^0]\!]_i, \ldots, [\![v^{2^{j-1}}]\!]_i$ and also

Algorithm 4: Computing $([\![v^2]\!], \ldots, [\![v^m]\!])$ from $[\![v]\!]$ in SHAREMIND

Data: $m \in \mathbb{N}$ and the value $[\![v]\!]$, where $v \in \mathbb{F}$, char $\mathbb{F} = 2$
Result: Values $[\![u_0]\!], \ldots, [\![u_m]\!]$, where $u_j = v^j$

$q \leftarrow \lceil \log \sqrt{m+1} \rceil$
$[\![u_0]\!] \leftarrow (1, 0, 0)$
$[\![u_1]\!] \leftarrow \mathsf{Reshare}([\![v]\!])$
Party P_i sends $[\![u_1]\!]_i$ to party P_{i+1}
for $j = 1$ **to** $2^{q-1} - 1$ **do**
    P_i computes $[\![u_{2j}]\!]_i \leftarrow [\![u_j]\!]_i^2$ and $[\![u_{2j}]\!]_{i-1} \leftarrow [\![u_j]\!]_{i-1}^2$
    P_i computes $[\![t]\!]_i \leftarrow [\![u_j]\!]_i \cdot [\![u_{j+1}]\!]_i + [\![u_j]\!]_i \cdot [\![u_{j+1}]\!]_{i-1} + [\![u_j]\!]_{i-1} \cdot [\![u_{j+1}]\!]_i$
    $[\![u_{2j+1}]\!] \leftarrow \mathsf{Reshare}([\![t]\!])$
    Party P_i sends $[\![u_{2j+1}]\!]_i$ to party P_{i+1}
foreach $j \in \{2^q, \ldots, m\}$ **do**
    Let $(r, s) \in \{0, \ldots, 2^q - 1\}$, such that $2^q r + s = j$
    Party P_i computes $[\![t]\!]_i \leftarrow [\![u_r]\!]_i^{2^q} \cdot [\![u_s]\!]_i + [\![u_r]\!]_i^{2^q} \cdot [\![u_s]\!]_{i-1} + [\![u_r]\!]_{i-1}^{2^q} \cdot [\![u_s]\!]_i$
    $[\![u_j]\!] \leftarrow \mathsf{Reshare}([\![t_j]\!])$

$[\![v^0]\!]_{i-1}, \ldots, [\![v^{2j-1}]\!]_{i-1}$. With these values, it can compute $[\![v^{2j}]\!]_i$ and $[\![v^{2j+1}]\!]_i$ (effectively, we are computing $v^{2j} = (v^j)^2$ and $v^{2j+1} = v^j \cdot v^{j+1}$). Party P_i can also compute $[\![v^{2j}]\!]_{i-1}$. It receives $[\![v^{2j+1}]\!]_{i-1}$ from P_{i-1}. In the second loop, we compute $v^j = (v^r)^{2^q} \cdot v^s$ for $j = 2^q \cdot r + s$. Again, $[\![(v^r)^{2^q}]\!]_i$ is locally computed from $[\![v^r]\!]_i$ by squaring it q times.

4.2 Speeding up the vector-only phase

A different kind of optimization is available if the ABB implementation is based on Shamir's secret sharing [32], using Gennaro et al.'s multiplication protocol [17] (examples are VIFF [12] and SEPIA [6]). Such ABB impelementations (for p parties), secure against t parties can be given if $2t + 1 \leq p$ (for passive security) or $3t + 1 \leq p$ (for active security). In such implementation, a value $v \in \mathbb{F}$ for a field $\mathbb{F}$ of size at least $p + 1$ is stored in the ABB as $[\![v]\!] = ([\![v]\!]_1, \ldots, [\![v]\!]_p)$, such that there exists a polynomial f over $\mathbb{F}$ with degree at most t and satisfying $f(0) = v$ and $f(c_i) = v_i$ for all $i \in \{1, \ldots, p\}$, where $\mathbf{C} = \{c_1, \ldots, c_p\}$ is a set of mutually different, public, fixed, nonzero elements of $\mathbb{F}$. The *share* v_i is kept by the i-th party P_i.

Our optimization relies on the computation of a scalar product $[\![\sum_{i=1}^k u_i v_i]\!]$ from the values $[\![u_1]\!], \ldots, [\![u_k]\!]$ and $[\![v_1]\!], \ldots [\![v_k]\!]$ stored inside the ABB having the same cost as performing a single multiplication of stored values. For reference, Alg. 5 presents the scalar product protocol in the SSS-based ABB providing passive security (thus $2t + 1 \leq p$) [17]. The multiplication protocol can be obtained from it simply by letting the length of the vectors to be 1. In this protocol, the values $\lambda_i^{\mathbf{C}}$ are the Lagrange interpolation coefficients satisfying $f(0) = \sum_{i=1}^p \lambda_i^{\mathbf{C}} f(c_i)$ for any polynomial f over $\mathbb{F}$ of degree at most $2t$. The

Algorithm 5: Scalar product protocol in an SSS-based ABB [17]

Data: Vectors $([\![u_1]\!], \ldots, [\![u_n]\!])$ and $([\![v_1]\!], \ldots, [\![v_n]\!])$
Result: Value $[\![w]\!]$, such that $w = \sum_{j=1}^{n} u_j v_j$
Party P_i computes $d_i \leftarrow \sum_{j=1}^{n} [\![u_j]\!]_i [\![v_j]\!]_i$
Party P_i picks a random polynomial f_i of degree at most t, such that $f_i(0) = d_i$.
Party P_i sends $f_i(c_j)$ to P_j
Party P_i computes $[\![w]\!]_i \leftarrow \sum_{j=1}^{p} \lambda_j^{\mathbf{C}} f_j(c_i)$.

Algorithm 6: Improved vector-only and online phases of the private lookup protocol

Data: Lagrange interpolation coefficients $\lambda_{j,k}^{\mathbf{I}}$
Data: Random non-zero $[\![r]\!]$ and its powers $[\![r^{-1}]\!], [\![r^2]\!], \ldots, [\![r^{m-1}]\!]$.
Data: Vector of values $([\![v_{i_1}]\!], \ldots, [\![v_{i_m}]\!])$ with $v_{i_1}, \ldots, v_{i_m} \in \mathbb{F}$.
Data: Index $[\![j]\!]$ to be looked up, with $j \in \{i_1, \ldots, i_m\}$.
Result: The looked up value $[\![w]\!] = [\![v_j]\!]$.
Vector-only phase
1 **foreach** $k \in \{0, \ldots, m-1\}$ **do** $[\![c_k]\!] \leftarrow \sum_{l=1}^{m} \lambda_{k,l}^{\mathbf{I}} [\![v_l]\!]$;
Online phase
2 $z \leftarrow \mathsf{retrieve}([\![j]\!] \cdot [\![r^{-1}]\!])$
3 **foreach** $j \in \{0, \ldots, m-1\}$ **do** $[\![\zeta_j]\!] \leftarrow z^j [\![r^j]\!]$;
4 $[\![w]\!] = ([\![c_0]\!], \ldots, [\![c_{m-1}]\!]) \cdot ([\![\zeta_0]\!], \ldots, [\![\zeta_{m-1}]\!])$

protocols providing active security are much more complex [10], but similarly have equal costs for multiplication and scalar product.

Our optimization consists of a reordering of the operations of the vector-only and online phases of the private lookup protocol, as depicted in Alg. 6. We see that compared to Alg. 1, we have moved the entire computation of the products $z^j [\![c_j]\!] [\![r^j]\!]$ to the online phase, thereby reducing the vector-only phase to the computation of certain linear combinations. The online phase becomes more complex, but only by a single scalar product, which costs the same as a single multiplication. The correctness and privacy arguments for Alg. 6 are the same as for Alg. 1.

Table 1. Communication costs (in elements of $\mathbb{F}$) of different private lookup protocols

Sharing	offline	vec-only	online
additive	$3\sqrt{m}$	$6m$	12
(public v)	$3\sqrt{m}$	0	12
Shamir's	$6m$	0	15
(public v)	$6m$	0	9

Unfortunately, we cannot use the optimizations of both Sec. 4.1 and Sec. 4.2 at the same time, as the cost of converting from one representation to the other would cancel any efficiency gains. If we have three parties and seek passive security against one of them, then our choices are given in Table. 1. Recall that multiplication in both representations and retrieval in the additive representation

requires the communication of 6 field elements in total. Retrieval in Shamir's secret sharing based representation requires 3 field elements to be sent.

5 Protocol for DFA execution

A DFA is a tuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q is a set of states, Σ is the alphabet (a set of characters), $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states and $\delta : Q \times \Sigma \to Q$ is the transition function. To *execute* a string $w = w_1 \cdots w_\ell \in \Sigma^*$ on A means to find the states $q_1, \ldots, q_\ell$, such that $q_i = \delta(q_{i-1}, w_i)$ for all $i \in \{1, \ldots, \ell\}$ and check whether $q_\ell \in F$.

In our implementation, the size of the problem — the numbers $|Q| = m$, $|\Sigma| = n$ and $|w| = \ell$ — is public, but δ and F are private. We need private lookup to implement δ. It is represented as a table of $|Q| \cdot |\Sigma|$ private values. We compute the private index from $[\![q_{i-1}]\!]$ and $[\![w_i]\!]$ and use it to find $[\![q_i]\!]$ from this table, seen as a vector of length $N = mn$. We have implemented DFA execution using both additive sharing and Shamir's sharing, in fields $GF(p)$ for $p = 2^{32} - 5$ and $GF(2^{32})$. We have measured the performance of Alg. 1 in all cases, as well as the optimizations of Alg. 4 and Alg. 6 in appropriate cases. Our tests were performed on three computing nodes, each of which was deployed on a separate machine. The computers in the cluster were connected by an Ethernet local area network with link speed of 1 Gbps. Each computer in the cluster had 48 GB of RAM and a 12-core 3 GHz CPU with Hyper Threading.

Our implementation is sub-optimal in the round complexity (which is $O(\ell)$), as it faithfully implements the definition of the DFA execution. Hence the running time of the online phase is currently dominated by the latency of the network. On the other hand, this also implies that many instances of DFA execution run in parallel would have almost the same runtime (for online phase) as the single instance. It is well-known that the DFA execution could be implemented in parallel fashion, using $O(\log \ell)$ time (or SMC rounds). This, however, increases the total work performed by the algorithm by a factor of $O(m)$.

We see that for the vector-only phase of the private lookup, we need the description of δ, but not yet the string w. This corresponds very well to certain envisioned cloud services, in particular to privacy-preserving spam filtering, where the spamminess is detected with regular expressions.

Table 2 presents the actual running times of our DFA execution implementation. All running times are in milliseconds, given with 2–3 significant digits. We have measured the running time for different automaton sizes m and alphabet sizes n, with $N = mn$ being between 6 and 30000. The length of the input string was always 2000 — the work performed by the algorithm, as well as its timing behavior is perfectly linear in this length.

We see that the running time of the online phase is indeed only slightly dependent on N: it is $\approx 286 + 0.042N$ µs per character of the input string when using the field $GF(p)$ (for $GF(2^{32})$, it is around $290 + 0.105N$). This slight dependence on N is caused by the local computations, the amount of which depends on N. Its effect would be even lower if the network latency were

Table 2. DFA execution benchmarks (times in milliseconds, $\ell = 2000$)

$(m,n) =$		$(3,2)$	$(15,10)$	$(100,30)$	$(1000,30)$
Using only Alg. 1 for lookup					
$GF(p)$, additive	offline	7	120	2260	23000
	vector-only	4	75	1440	22000
	online	560	590	830	3100
$GF(2^{32})$, additive	offline	11	160	3000	30000
	vector-only	5	110	2200	49000
	online	563	620	1230	6800
$GF(p)$, Shamir	offline	7	120	2600	27000
	vector-only	4	86	1520	23000
	online	580	580	810	3100
$GF(2^{32})$, Shamir	offline	12	200	3900	38000
	vector-only	6	128	2400	53000
	online	570	620	1190	6800
With optimizations of Alg. 4 or Alg. 6					
$GF(2^{32})$, additive	offline	12	72	1140	10600
$GF(p)$, Shamir	vector-only	0	1	89	8100
	online	900	900	1230	4300
$GF(2^{32})$, Shamir	vector-only	0	2	310	32000
	online	900	940	1900	13000

higher. Other phases depend much more on N: offline phase requires $0.4N$ µs and vector-only phase $0.23N + 4.7 \cdot 10^{-6}N^2$ µs per character for $GF(p)$.

Among related work, running times of their algorithm implementations have been presented in [15, 30]. Our implementation is significantly more efficient than [15]. They report the running time of 8 seconds for processing a string of length $\ell = 10$ on an automaton with $mn = 40000$. Compare this number with our reported running time of ≈ 3 seconds for the online phase or even with ≈ 40 seconds for all three phases of processing a 2000-character string on an automaton with $mn = 30000$.

Our running times do not seem that impressive when compared to [30], where, with a more optimized implementation inspired by garbled circuits, running times as low as 12 seconds are reported for $n = 2$ and $\{m, \ell\} = \{20, 150000\}$. But even then, if we consider $mn\ell$ to be a valid measure of the size of the problem, our implementation is a couple of times faster (and the online phase requires only 10% of that time). Also, they are solving a narrower problem, with one of the parties knowing the automaton and the other knowing the input string, while our protocols are universally composable. On the other hand, their implementation is secure against malicious adversaries, while we have tested our protocols only on ABB implementations secure against passive attacks only.

Table 3. SSSD execution benchmarks (times in seconds)

n	100	100	100	100	300	300	300	600	600	600	1000	1000	2000
m	100	600	1000	9900	300	1800	3000	600	3600	6000	1000	6000	2000
offline	0.3	1.3	1.9	19	5.2	31	52	41	240	400	190	1100	1540
online	6.0	7.9	9.2	68	10.4	49	72	39	190	310	110	580	550

6 Protocol for SSSD

Let $G = (V, E)$ be a directed graph with $s, t : E \to V$ giving the source and target, and $w : E \to \mathbb{N}$ giving the length of each edge. Let $v_0 \in V$. Bellman-Ford (BF) algorithm for SSSD starts by defining $d_0[v] = 0$, if $v = v_0$, and $d_0[v] = \infty$ for $v \in V \backslash \{v_0\}$. It will then compute $d_{i+1}[v] = \min(d_i[v], \min_{e \in t^{-1}(v)} d_i[s(e)] + w(e))$ for all $v \in V$ and $i \in \{0, \ldots, |V| - 2\}$. The vector $d_{|V|-1}$ is the result of the algorithm.

We have implemented the BF algorithm on top of the Sharemind platform, hiding the structure of the graph, as well as the lengths of edges. In our implementation, the numbers $n = |V|$ and $m = |E|$ are public, and so are the in-degrees of vertices (obviously, these could be hidden by using suitable paddings). In effect, the mapping t in the definition of the graph is public, while the mappings s and w are private. During the execution, we use private lookup to find $d_i[s(e)]$. As the vectors d_i have to be computed one after another, but the elements of the same vector can be computed in parallel, our implementation has $O(n)$ rounds in the online phase.

As the vector d_i is not yet available at the start of computation, we use the optimized vector-only phase to avoid an $O(n)$ factor during the execution of the BF algorithm. Hence we use Shamir's secret sharing based ABB implementation. We have to perform arithmetic and comparisons with secret values, hence we must use a prime field as the field $\mathbb{F}$ (we use $GF(p)$ with $p = 2^{32} - 5$).

Table 3 presents the actual running times of our implementation of the Bellman-Ford algorithm on sparse graphs. All running times are in seconds, given with 2–3 significant digits. We have measured the running time for different graphs with n vertices (where $100 \leq n \leq 2000$) and m edges, also matching the problem sizes in related work. Hence we have included cycle graphs (where $m = n$), as well as the complete directed graph on 100 vertices (with $m = 9900$). We also believe that in applications, the used graphs are often planar. Thus we have selected the parameters of certain graphs to match planar graphs, where most faces are triangles and edges are bidirectional ($m \approx 6n$).

We see that in our tests, the offline phase requires around $4.74n^2 + 1.28mn + 0.181mn^2$ µs, and the online phase around $14.7n^2 + 67.3mn + 0.0257mn^2$ µs. The asymptotic running time of the BF algorithm is $O(mn)$. Hence we see that the online phase depends much less on the mn^2 term than the offline phase.

The running times reported in related work are much higher. The protocols of [1] are implemented on VIFF [12], running with three parties on a single

machine, and requiring 5622 s for SSSD in 128-vertex complete graph. Compare this with our running time of 87 s (offline+online) for the 100-vertex complete graph. We require (550+1540) s (online+offline) for a graph with $m = n = 2000$. In [22], the same graph requires around 10000 s with implementation based on SPDZ [23] (hence their time probably does not include SPDZ precomputations).

7 Conclusions

In this paper, we have shown that arithmetic black boxes support fast lookups from private tables according to a private index. We have used this operation to obtain very efficient algorithms for certain tasks. Our results show that for private lookups in an ABB, complex techniques based on Oblivious RAMs [13] are not necessary. Beside the DFA execution or the Bellman-Ford algorithms, we expect our techniques to have wide applicability in making algorithms with sensitive data reading patterns privacy preserving.

References

1. Aly, A., Cuvelier, E., Mawet, S., Pereira, O., Vyve, M.V.: Securely solving simple combinatorial graph problems. In: Sadeghi, A.R. (ed.) Financial Cryptography. Lecture Notes in Computer Science, vol. 7859, pp. 239–257. Springer (2013)
2. Blanton, M., Aliasgari, M.: Secure Outsourcing of DNA Searching via Finite Automata. In: Foresti, S., Jajodia, S. (eds.) DBSec. Lecture Notes in Computer Science, vol. 6166, pp. 49–64. Springer (2010)
3. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: Jajodia, S., López, J. (eds.) ESORICS. Lecture Notes in Computer Science, vol. 5283, pp. 192–206. Springer (2008)
4. Bogdanov, D., Niitsoo, M., Toft, T., Willemson, J.: High-performance secure multiparty computation for data mining applications. Int. J. Inf. Sec. 11(6), 403–418 (2012)
5. Brickell, J., Shmatikov, V.: Privacy-preserving graph algorithms in the semi-honest model. In: Roy, B.K. (ed.) ASIACRYPT. Lecture Notes in Computer Science, vol. 3788, pp. 236–252. Springer (2005)
6. Burkhart, M., Strasser, M., Many, D., Dimitropoulos, X.: SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In: USENIX Security Symposium. pp. 223–239. Washington, DC, USA (2010)
7. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: FOCS. pp. 136–145. IEEE Computer Society (2001)
8. Catrina, O., Saxena, A.: Secure computation with fixed-point numbers. In: Sion, R. (ed.) Financial Cryptography and Data Security. LNCS, vol. 6052, pp. 35–50. Springer (2010)
9. Cramer, R., Damgård, I., Nielsen, J.B.: Multiparty computation from threshold homomorphic encryption. In: Pfitzmann, B. (ed.) EUROCRYPT. Lecture Notes in Computer Science, vol. 2045, pp. 280–299. Springer (2001)
10. Cramer, R., Damgrd, I.: Multiparty computation, an introduction. In: Contemporary Cryptology, pp. 41–87. Advanced Courses in Mathematics - CRM Barcelona, Birkhuser Basel (2005), http://dx.doi.org/10.1007/3-7643-7394-6_2

11. Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J.B., Toft, T.: Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In: Halevi, S., Rabin, T. (eds.) TCC. Lecture Notes in Computer Science, vol. 3876, pp. 285–304. Springer (2006)
12. Damgård, I., Geisler, M., Krøigaard, M., Nielsen, J.B.: Asynchronous Multiparty Computation: Theory and Implementation. In: Jarecki, S., Tsudik, G. (eds.) Public Key Cryptography. Lecture Notes in Computer Science, vol. 5443, pp. 160–179. Springer (2009)
13. Damgård, I., Meldgaard, S., Nielsen, J.B.: Perfectly secure oblivious ram without random oracles. In: Ishai, Y. (ed.) TCC. Lecture Notes in Computer Science, vol. 6597, pp. 144–163. Springer (2011)
14. Damgård, I., Nielsen, J.B.: Universally composable efficient multiparty computation from threshold homomorphic encryption. In: Boneh, D. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 2729, pp. 247–264. Springer (2003)
15. Frikken, K.B.: Practical Private DNA String Searching and Matching through Efficient Oblivious Automata Evaluation. In: Gudes, E., Vaidya, J. (eds.) DBSec. Lecture Notes in Computer Science, vol. 5645, pp. 81–94. Springer (2009)
16. Gennaro, R., Hazay, C., Sorensen, J.S.: Text search protocols with simulation based security. In: Nguyen, P.Q., Pointcheval, D. (eds.) Public Key Cryptography. Lecture Notes in Computer Science, vol. 6056, pp. 332–350. Springer (2010)
17. Gennaro, R., Rabin, M.O., Rabin, T.: Simplified vss and fact-track multiparty computations with applications to threshold cryptography. In: PODC. pp. 101–111 (1998)
18. Gentry, C., Goldman, K.A., Halevi, S., Jutla, C.S., Raykova, M., Wichs, D.: Optimizing oram and using it efficiently for secure computation. In: Cristofaro, E.D., Wright, M. (eds.) Privacy Enhancing Technologies. Lecture Notes in Computer Science, vol. 7981, pp. 1–18. Springer (2013)
19. Goldreich, O., Micali, S., Wigderson, A.: How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In: STOC. pp. 218–229. ACM (1987)
20. Goldreich, O., Ostrovsky, R.: Software Protection and Simulation on Oblivious RAMs. J. ACM 43(3), 431–473 (1996)
21. Henecka, W., Kögl, S., Sadeghi, A.R., Schneider, T., Wehrenberg, I.: TASTY: tool for automating secure two-party computations. In: CCS '10: Proceedings of the 17th ACM conference on Computer and communications security. pp. 451–462. ACM, New York, NY, USA (2010)
22. Keller, M., Scholl, P.: Efficient, Oblivious Data Structures for MPC. Cryptology ePrint Archive, Report 2014/137 (2014), http://eprint.iacr.org/
23. Keller, M., Scholl, P., Smart, N.P.: An architecture for practical actively secure mpc with dishonest majority. In: Sadeghi et al. [31], pp. 549–560
24. Kushilevitz, E., Ostrovsky, R.: Replication is NOT Needed: SINGLE Database, Computationally-Private Information Retrieval. In: FOCS. pp. 364–373. IEEE Computer Society (1997)
25. Launchbury, J., Diatchki, I.S., DuBuisson, T., Adams-Moran, A.: Efficient lookup-table protocol in secure multiparty computation. In: Thiemann, P., Findler, R.B. (eds.) ICFP. pp. 189–200. ACM (2012)
26. Lipmaa, H.: First CPIR Protocol with Data-Dependent Computation. In: Lee, D., Hong, S. (eds.) ICISC. Lecture Notes in Computer Science, vol. 5984, pp. 193–210. Springer (2009)

27. Lipmaa, H., Toft, T.: Secure equality and greater-than tests with sublinear online complexity. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M.Z., Peleg, D. (eds.) ICALP (2). Lecture Notes in Computer Science, vol. 7966, pp. 645–656. Springer (2013)
28. Liu, C., Huang, Y., Shi, E., Katz, J., Hicks, M.: Automating Efficient RAM-Model Secure Computation. In: Proceedings of 2014 IEEE Symposium on Security and Privacy. IEEE (2014)
29. Malka, L., Katz, J.: Vmcrypt - modular software architecture for scalable secure computation. Cryptology ePrint Archive, Report 2010/584 (2010), http://eprint.iacr.org/
30. Mohassel, P., Niksefat, S., Sadeghian, S.S., Sadeghiyan, B.: An Efficient Protocol for Oblivious DFA Evaluation and Applications. In: Dunkelman, O. (ed.) CT-RSA. Lecture Notes in Computer Science, vol. 7178, pp. 398–415. Springer (2012)
31. Sadeghi, A.R., Gligor, V.D., Yung, M. (eds.): 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013. ACM (2013)
32. Shamir, A.: How to share a secret. Commun. ACM 22(11), 612–613 (1979)
33. Stefanov, E., van Dijk, M., Shi, E., Fletcher, C.W., Ren, L., Yu, X., Devadas, S.: Path ORAM: an extremely simple oblivious RAM protocol. In: Sadeghi et al. [31], pp. 299–310
34. Troncoso-Pastoriza, J.R., Katzenbeisser, S., Celik, M.U.: Privacy preserving error resilient DNA searching through oblivious automata. In: Ning, P., di Vimercati, S.D.C., Syverson, P.F. (eds.) ACM Conference on Computer and Communications Security. pp. 519–528. ACM (2007)
35. Wei, L., Reiter, M.K.: Third-Party Private DFA Evaluation on Encrypted Files in the Cloud. In: Foresti, S., Yung, M., Martinelli, F. (eds.) ESORICS. Lecture Notes in Computer Science, vol. 7459, pp. 523–540. Springer (2012)
36. Yao, A.C.C.: Protocols for secure computations (extended abstract). In: FOCS. pp. 160–164. IEEE (1982)